InfraRed Documentation

Release IR-stable-v1.0.0.0.0-1

yfried

Apr 20, 2017

Contents

CHAPTER 1

Contents:

Introduction

InfraRed is tool used for automated deployments of various OpenStack environments. It does not try to be focused on CI use-cases only, it is focused on automation in general. It is written in Python 2.7 and using Ansible as deployment backend. Python dependencies are handled by pip package manager.

Workflow is divided into 3 separated steps:

1. Provisioning (ir-provisioner tool)
2. Installation (ir-installer tool)
3. Testing (ir-tester tool)

Please see Setup page first or proceed to guide for impatient (Quickstart).

Quickstart

Note: This guide assumes:

- Latest version of Python 2 installed
- Virtualenv is used
- Prerequisites are set-up
- We strongly urge to read all Setup instructions first
- Quickstart is assuming you are use Fedora as main distro for deployment and provisioning (RHEL needs private adjustments)

Clone InfraRed stable from GitHub:

```
git clone https://github.com/rhosqeauto/InfraRed.git -b stable
```

Note: This is documentation for stable version. Check in top left corner of this page if your stable branch tag matches version of documentation. If not true, let us know!

Install from source using pip:

```
cd InfraRed
pip install --upgrade pip setuptools
pip install .
cp ansible.cfg.example ansible.cfg
```

Warning: While most topologies will work 'out of the box', some topologies (like external ceph, netapp, etc) requires internal credentials which we cannot upload upstream. Users with access to redhat internal network can run the following command to download a file contains some credentials & other sensitive data, other user will have to provide this data explicitly everywhere there is a reference to private variables.

```
wget --no-check-certificate https://url.corp.redhat.com/infrared-private -O␣
↪infrared-private.yml
```

Basic Usage Example

Provisioning

In this example we'll use virsh provisioner in order to demonstrate how easy and fast it is to provision machines using InfraRed. For basic execution, the user should only provide data for the mandatory parameters, this can be done by two ways:

1. *CLI*
2. *INI File*

CLI

To list all parameters (for *virsh*) and their description, run:

```
ir-provisioner virsh --help
```

Notice that the only three mandatory paramters in *virsh* provisioner are:

- `--host-address` - the host IP or FQDN to ssh to
- `--host-key` - the private key file used to authenticate to your `host-address` server
- `--topology-nodes` - type and role of nodes you would like to deploy (e.g: controller:3 == 3 VMs that will act as controllers)

We can now execute the provisioning process by providing those parameters through the CLI:

```
ir-provisioner virsh --host-address=$HOST --host-key=$HOST_KEY --topology-nodes=
↪"undercloud:1,controller:1,compute:1" -e @infrared-private.yml
```

Note: The value of the topology-nodes option is a comma-separated string in a "type:amount" format. Please check the settings/topology dir for a complete list of the available types. (In the example above, three nodes will be provisioned: 1 undercloud, 1 controller & 1 compute)

That is it, the machines are now provisioned and accessible.

Note: You can also use the auto-generated ssh config file to easily access the machines

```
ssh -F ansible.ssh.config controller-0
```

INI File

Unlike with CLI, here a new configuration file (INI based) will be created. This file contains all the default & mandatory parameters in a section of its own (named 'virsh' in our case), so the user can easily replace all mandatory parameters. When the file is ready, it should be provided as an input for the '–from-file' option.

Generate INI file for *virsh* provisioner:

```
ir-provisioner virsh --generate-conf-file virsh_prov.ini
```

Review the config file and edit as required:

Listing 1.1: virsh_prov.ini

```
[virsh]
host-key = Required argument. Edit with any value, OR override with CLI: --host-key=
↪<option>
host-address = Required argument. Edit with any value, OR override with CLI: --host-
↪address=<option>
topology-nodes = Required argument. Edit with one of the allowed values OR override␣
↪with CLI: --topology-nodes=<option>
topology-network = default.yml
host-user = root
```

Note: `host-key`, `host-address` and `topology-nodes` don't have default values. All arguments can be edited in file or overridden directly from CLI.

Note: Do not use double quotes or apostrophes for the string values in the configuration ini file. Infrared will NOT remove those quotation marks that surround the values.

Edit mandatory parameters values in the INI file:

```
[virsh]
host-key = ~/.ssh/id_rsa
host-address = my.host.address
topology-nodes = undercloud:1,controller:1,compute:1
topology-network = default.yml
host-user = root
```

Execute provisioning using the newly created INI file:

```
ir-provisioner virsh --from-file=virsh_prov.ini -e @infrared-private.yml
```

Note: You can always overwrite parameters from INI file with parameters from CLI

```
ir-provisioner virsh --from-file=virsh_prov.ini --topology-nodes="undercloud:1,
↪controller:1,compute:1,ceph:1" -e @infrared-private.yml
```

Done. Quick & Easy!

Warning: Users without access to redhat internal network will have to provide a url to a guest image using the "–image-url" option

Installing

Now let's demonstrate the installation process by deploy an OpenStack environment using redhat OSPD (OpenStack Director) on the nodes we have provisioned in the previous stage (The deployment in this case will be 'virthost' type, see how to setup Virthost machine).

Just like in the provisioning stage, here also the user should take care of the mandatory parameters (by CLI or INI file) in order to be able to start the installation process. Lets provide the mandatory parameter (`deployment-files`) and choose to work with RHOS version 8, this time using the CLI only:

```
ir-installer ospd --deployment-files=$PWD/settings/installer/ospd/deployment/virt --
↪product-version=8 --product-core-version=8 -e @infrared-private.yml
```

Note: Please notice that the `deployment-file` parameters requires a full path of the deployment files dir.

Done.

OSPD Quickstart

InfraRed provides a quick solution to deploy OSPD with a pre-configured undercloud from latest build for testing/POC.

1. Provision: No undercloud node should be provisioned in the provisioning stage.

```
ir-provisioner virsh --host-address=$HOST --host-key=$HOST_KEY --topology-nodes=
↪"controller:1,compute:1" -e @infrared-private.yml
```

2. Install: InfraRed will notice that no UC is provided and will build one from a snapshot of an installed UC from latest available build.

```
ir-installer ospd --deployment-files=$PWD/settings/installer/ospd/deployment/virt --
↪product-version=9 --product-core-version=9 -e @infrared-private.yml
```

For detailed information on the usage of the various installers, provisioners & tester continue to Using InfraRed

Setup

Supported distros

Currently supported distros are:

- Fedora 22, 23
- RHEL 7.2 (best effort only, deprecated)
- RHEL 7.3

Warning: Python 2.7 and virtualenv are required.

Prerequisites

Warning: Sudo or root access is needed to install prerequisities!

General requirements:

```
sudo dnf/yum install git gcc libffi-devel openssl-devel sshpass
```

Note: Dependencies explained:

- git - version control of this project
- gcc - used for compilation of C backends for various libraries
- libffi-devel - required by cffi
- openssl-devel - required by cryptography
- sshpass - required by wait_for ansible module

Closed Virtualenv is required to create clean python environment separated from system:

```
sudo dnf/yum install python-virtualenv
```

Ansible requires python binding for SELinux:

```
sudo dnf/yum install libselinux-python
```

otherwise it won't be able to run modules with copy/file/template functions!

Note: libselinux-python is in *Prerequisites* but doesn't have a pip package. It must be installed on system level.

Warning: Ansible requires also **libselinux-python** installed on all nodes using copy/file/template functions. Without this step all such tasks will fail!

Virtualenv

InfraRed shares many dependencies with other OpenStack products and projects. Therefore there's a high probability of conflicts with python dependencies, which would result either with InfraRed failure, or worse, with breaking dependencies for other OpenStack products. When working from source, it is recommended to use python virtualenv to avoid corrupting the system packages:

```
virtualenv .venv
source .venv/bin/activate
```

Warning: It is mandatory that latest pip is used (especially in when working with RHEL)!

```
pip install --upgrade pip setuptools
```

Note: On Fedora 23 with EPEL repository enabled, RHBZ#1103566 also requires:

```
dnf install redhat-rpm-config
```

Installation

Clone stable branch from Github repository:

```
git clone https://github.com/rhosqeauto/InfraRed.git -b stable
```

Install InfraRed from source:

```
cd InfraRed
pip install .
```

Note: For development work it's better to install in editable mode and work with master branch

```
git checkout master
pip install -e .
```

Configuration

Note: InfraRed only requires explicit configuraion file when non-default values are used.

InfraRed will look for `infrared.cfg` in the following order:

1. Environment variable: `$IR_CONFIG=/my/config/infrared.cfg`
2. In working directory: `./infrared.cfg`
3. In user home directory: `~/.infrared.cfg`
4. In system settings: `/etc/infrared/infrared.cfg`

If no configuration file is supplied, InfraRed will load default values as listed in ‘‘infrared.cfg.example

Set up ansible config if it was not configured already:

```
cp ansible.cfg.example ansible.cfg
```

Additional settings

In InfraRed configuration file, you can adjust where ansible looks for directories and entry/cleanup playbooks:

Listing 1.2: infrared.cfg.example

```
InfraRed configuration file
# ============================

[defaults]
settings  = settings
modules   = library
roles     = roles
playbooks = playbooks

[provisioner]
main_playbook = provision.yml
cleanup_playbook = cleanup.yml

[installer]
main_playbook = install.yml
cleanup_playbook = cleanup.yml

[tester]
main_playbook = test.yml
cleanup_playbook = cleanup.yml
```

Private settings

Infrared allows user to define several folders to store settings and spec files. This can be used, for example, to store public and private settings separately. To define additional settings folders edit the `settings` option in the Infrared configuration file:

```
[defaults]
settings  = settings:private_settings
...
```

Note: InfraRed tool must be tied to infrastructure at certain level, therefore requires part of configuration not shared publicly. It is assumed this part will be located in private settings.

For more questions please contact us.

Virthost machine

Virthost machine is the target machine where InfraRed’s virsh provisioner will create virtual machines and networks (using libvirt) to emulate baremetal infrastructure.

As such there are few specific requirements it has to meet.

Generally, It needs to have **enough memory and disk** storage to hold multiple decent VMs (each with GBytes of RAM and dozens of GB of disk). Also for acceptable responsiveness (speed of deployment/testing) just <4 threads or low GHz CPU is not a recommended choice (if you have old and weaker CPU than current mid-high end mobile phone CPU you may suffer performance wise - and so more timeouts during deployment or in tests).

Especially, for Ironic (TripleO) to control them, those **libvirt VMs** need to be bootable/controllable for **iPXE provisioning**. And also extra user has to exist, which can ssh in the virthost and control (restart...) libvirt VMs.

Note: InfraRed is currently attempting to configure or validate all (most) of this but it's scattered across all provisiner/installer steps. Due to nature of installers such as OSPd and current InfraRed structure it may not be 100% safe for rerunning (failure in previous run may prevent following one from succeeding in these preparation steps). We are currently working on a more idempotent approach which should resolve the above issues (if present).

What **user has to provide**:

- have machine with **sudoer user ssh access** and **enough resources**, as minimum requirements for one VM are:
  - VCPU: 2|4|8
  - RAM: 8|16
  - HDD: 40GB+
  - in practice disk may be smaller, as they are thin provisioned, as long as you don't force writing all the data (aka Tempest with rhel-guest instead of cirros etc)
- tested is just **RHEL-7.3** as OS, with also **CentOS** expected to work
  - may work with other distributions (best-effort/limited support)
- **yum repositories** has to be **preconfigured** by user (foreman/...) before using InfraRed so it can install dependencies
  - esp. for InfraRed to handle `ipxe-roms-qemu` it requires either **RHEL-7.3-server channel**, or (deprecated) *RHEL-7.2 with OSP<10 channels* (10+ is 7.3)

What **InfraRed takes care of**:

- `ipxe-roms-qemu` package of at least `version 2016xxyy` needs to be installed
- other basic packages installed
  - `libvirt`, `libguestfs{-tools,-xfs}`, `qemu-kvm`, `wget`, `virt-install`
  - `virt-manager`, `xorg-x11-apps`, `xauth`, `virt-viewer` possibly for debugging (or multiple ssh tunnels can be used)
- **virtualization support** (VT-x/AMD-V)
  - ideally with **nested=1** support
- `stack` user created with polkit privileges for *org.libvirt.unix.manage*
- **ssh key** with which InfraRed can authenticate (created and) added for *root* and *stack* user, atm they are handled differently/separately:
  - for *root* the `infared/id_rsa.pub` gets added to authorized_keys
  - for *stack* `infrared/id_rsa_undercloud.pub` is added to authorized_keys, created/added later during installation

Using InfraRed

General workflow

InfraRed framework is divided into three logically separated stages (tools):

- `ir-provisioner`
- `ir-installer`
- `ir-tester`

You can get general usage information with the `--help` option:

```
ir-<stage> --help
```

Output will display supported options you can pass to `ir-<stage>`, as well as available positional arguments for current stage (e.g. for provisioner these are foreman, virsh, openstack, ...):

Also, you can invoke help for specific positional arguments (supported provisioners, in this case):

```
ir-<stage> virsh --help
```

Note: Positional arguments are generated dynamically from spec files - order and amount might change in time.

Note: Stages are physically separated, you can execute them in mixed (but meaningful) order. Example:

```
ir-provisioner virsh
ir-installer ospd
ir-tester tempest
ir-installer ospd --scale
ir-tester tempest
```

Currently, executing different sub-commands of the same stage (i.e. `ir-provisioner beaker` and then `ir-provisioner virsh`) is possible but the user must save the created inventory files (`hosts-provisioner`) between exectuions as they will overwrite each other

---

Passing parameters

Note: By nature of the project, many configurable details like passwords, keys, certifcates, etc... cannot be stored in a public GitHub repo. We keep a private repo for internal Red Hat users that mirrors the `settings tree`. Using the Multi-settings feature in `infrared.cfg` file, InfraRed will search those directories for files missing from the public settings directory.

InfraRed expects that selected workflow (playbook and roles) will be provided with all mandatory parameters. There are several ways to do it:

- Use separate private configuration directory
- Include standalone file(s) containing additional (private) settings as explicit input file(s) (`-i` or `--input` parameters), for example:

```
ir-<stage> --input private.yml
```

Listing 1.3: private.yml

```
---
private:
   provisioner:
       beaker:
       base_url: "https://beaker_instance_url/"
       username: "..."
       password: "..."
...
```

- Use command line `ir-<stage> --param1 --param2 ...`

Note: Best practice is store infrastructure-specific configuration file(s) in private repository and fetch such file(s) before deployment.

Provisioners

For list of supported provisioners invoke:

```
$ ir-provisioner [<prov_name>] --help|-h
```

Beaker

Entry point:

```
playbooks/provisioner/beaker/main.yml
```

Beaker provisioner is designed to work with instances of Beaker project at least version 22.3. It is based custom ansible module built on top of

```
library/beaker_provisioner.py
```

script. While Beaker can support working with Kerberos, the usage is still limited, therefore authentication is done using XML-RPC API with credentials for dedicated user.

See appropriate value of `ssh_pass` for your `beaker_username` in *Website -> Account -> Preferences -> Root Password* if you didn't setup one. For proper XML-RPC calls `cert_file` must be provided.

Also, for each run you will need to set proper node-specific values:

```
...
Beaker system:
  --fqdn FQDN                Fully qualified domain name of a system
  --distro-tree DISTRO-TREE  Distro Tree ID Default value: 71576
...
```

Foreman

Entry point:

```
playbooks/provisioner/foreman/main.yml
```

Warning: Currently, Foreman provisioning supports only the ability to rebuild hosts (without the option change the operating system):

```
ir-provisioner [...] foreman [...]
```

Foreman provisioner is designed to work with instances of Foreman project at least version 1.6.3. It is based custom ansible module built on top of

```
library/foreman_provisioner.py
```

Foreman provisioner expects that provisioned node has configured relevant puppet recipies to provide basic SSH access after provisioning is done.

To get more details on how to provision hosts using Foreman:

```
$ ir-provisioner foreman --help
```

Openstack

Entry point:

```
playbooks/provisioner/openstack/main.yml
```

Provisioner is designed to work with existing instances of OpenStack. It is based on native ansible's cloud modules. Workflow can be separated into following stages:

- Create network infrastructure
- Create instance of virtual machine and connect to network infrastructure
- Wait until instance is booted and reachable using SSH

Note: Openstack provisioner is tested against Kilo version.

InfraRed interacts with cloud using os-client-config library. This library expects properly configured cloud.yml file in filesystem, however it is possible to position this file in InfraRed's directory.

Listing 1.4: clouds.yml

```
clouds:
    cloud_name:
        auth_url: http://openstack_instance:5000/v2.0
        username: <username>
        password: <password>
        project_name: <project_name>
```

`cloud_name` can be then referenced with `--cloud` parameter provided to `ir-provisioner`:

```
ir-provisioner ... --cloud cloud_name ...
```

Note: You can also ommit the cloud parameter, then InfraRed expects you alredy sourced keystonerc of targeted cloud:

```
source keystonerc
ir-provisioner openstack ...
```

Last important parameter is `--dns` which must be set to point to local DNS server in your infrastructure.

Virsh

Entry point:

```
playbooks/provisioner/virsh/main.yml
```

Virsh provisioner is explicitly designed to be used for setup of virtual OpenStack environments. Such environments are used to emulate production environment of OpenStack director instances on one baremetal machine. It requires prepared baremetal host to be reachable through SSH initially. Topology created using virsh provisioner is called "virthost".

First, Libvirt and KVM environment is installed and configured to provide virtualized environment. Then, virtual machines are created for all requested nodes. These VM's are used in OSPd installer as undercloud, overcloud and auxiliary nodes.

Please see Quickstart guide where usage is demonstrated.

Cleanup

virsh cleanup will discover virsh nodes and networks on the host and delete them as well as their matching disks. To avoid cleanup of specific nodes/networks use extra vars `ignore_virsh_nodes` and `ignore_virsh_nets`:

```
ir-provisioner [...] virsh [...]  --cleanup \
    --host-address=example1.redhat.com \
    --host-key=~/.ssh/id_rsa \
    --extra-vars ignore_virsh_nodes=MY-NODE-0 \
    --extra-vars ignore_virsh_nets=MY-PERSISTENT-NETWORK
```

By default, cleanup will only ignore `default` network (automatically created by *libvirt*). Overriding the `ignore_virsh_nets` variable will delete this network unless explicitly specified

Warning: Arguments like `images` and `topology` are required by cleanup even though they are never used. This will be fixed in future versions.

Warning: Cleanup won't install libvirt packages and requirements. If *libvirtd* service is unavailable, cleanup be skipped

Network layout

Baremetal machine used as host for such setup is called Virthost. The whole deployment is designed to work within boundaries of this machine and (except public/natted traffic) shouldn't reach beyond. Following layout is part of

default setup defined in default.yml. User can also provide his own network layout (example network-sample.yml).

```
      Virthost
          |
          +---------+ nic0 - public IP
          |
          +---------+ nic1 - not managed
          |
            ...                                                 Libvirt VM's
          |                                                           |
    ------+---------+ data bridge (ctlplane, 192.0.2/24)              +-------+ data␣
↪(nic0)
    |     |                                                           |
libvirt --+---------+ management bridge (nat, dhcp, 172.16.0/24)     +-------+␣
↪managementnt (nic1)
    |     |                                                           |
    ------+---------+ external bridge (nat, dhcp, 10.0.0/24)         +-------+ external␣
↪(nic2)
```

On virthost, there are 3 new bridges created with libvirt - data, management and external. Most important is data network which does not have dhcp and nat enabled. This network is used as ctlplane for OSP director deployments (OSPd installer). Other (usually physical) interfaces are not used (nic0, nic1, ...) except for public/natted traffic. External network is used for SSH forwarding so client (or ansible) can access dynamically created nodes.

Virsh provisioner workflow:

1. Setup libvirt and kvm environment
2. Setup libvirt networks
3. Download base image for undercloud (`--image`)
4. Create desired amount of images and integrate to libvirt
5. Define virtual machines with requested parameters (`--topology-nodes`)
6. Start virtual machines

Environments prepared such way are usually used as basic virtual infrastructure for OSPd installer.

Note: Virsh provisioner has currently idempotency issues, therefore `ir-provisioner virsh ... --cleanup` must be run before reprovisioning every time.

Custom images

If you need to provide your own prepared images for virsh provisioner, you can use handy feature overriding "import_url" option:

```
ir-provisioner ... \
-e topology.nodes.<node name>.disks.disk1.import_url=http://.../image.qcow2 ... \
...
```

Installers

For list of supported installers invoke:

```
$ ir-installer [<installer_name>] --help|-h
```

Packstack

Entry point:

```
playbooks/installer/packstack/main.yml
```

Infrared allows to use Packstack installer to install OpenStack:

```
$ ir-installer --inventory hosts packstack --product-version=8
```

Required arguments are:

- `--product-version` - the product version to install

Settings structure

The path for the main settings file for packstack installer:

```
settings/installer/packstack/packstack.yml
```

This file provides defaults settings and default configuration options for various packstack answer files. Additional answer options can be added using the the following approaches:

- Using a non default config argument value:

  ```
  $ ... --config=basic_neutron.yml
  ```

- Using the extra-vars flags:

  ```
  $ ... --product-version=8 --extra-vars=installer.config.CONFIG_DEBUG_MODE=no
  ```

- Network based answer file options can be selected whether by choosing network backend or by modyfing config with –extra-vars:

  ```
  $ ... --product-version=8 --network=neutron.yml --netwrok-variant=neutron_gre.yml

  $ ... --product-version=8 --network=neutron.yml --netwrok-variant=neutron_gre.yml␣
  ↪\
        --extra-vars=installer.network.config.CONFIG_NEUTRON_USE_NAMESPACES=n
  ```

Both *installer.network.config.** and *installer.config.** options will be merged into one config and used as the answer file for Packstack.

OpenStack director

Entry point:

```
playbooks/installer/ospd/main.yml
```

OSPd deployment in general consists of following steps:

- Undercloud deployment

- Virthost tweaks
- Image management
- Introspection
- Flavor setup
- Overcloud deployment

You can find full documentation at Red Hat OpenStack director.

There are 2 OSPd deployment types currently supported. The API is the same but different input is required and different assumptions are made for each deployment type:

- Baremetal (BM)

  Normal deployment of openstack where all nodes are physical hosts.

  Users need to provide:

  - `--deployment-files` - directory with various files and templates, describing the OverCloud (such as `instackenv.json`).
  - `--undercloud-config` - `undercloud.conf` file. If not provided, the sample configuration file will be used.
  - `--instackenv-file` - `instackenv.json` file.

  Both paths must be absolute paths:

```
ir-installer ospd [...] --deployment-files=/absolute/path/to/templates/directory␣
↪[...] --undercloud-config=/home/myuser/undercloud.conf
```

  The details of such directory can be found under settings tree

- Virthost (VH)

  Using *virsh* provisioner, deploy openstack on virtual machines hosted on a single hypervisor (aka Virthost).

  This is a common use-case for POC, development and testing, where hardware is limited. OSPD requires special customization to be nested on OpenStack clouds, so using local virsh VMs is a common solution.

  Expects the following network deployment (created by the `virsh` provisioner):

  nic1 - data

  - Referred to as "ctlplane" by OSPd documentation
  - Does not have dhcp and nat enabled (OSPd will later take dhcp/nat ownership for this network)
  - Used by OSPD to handle dhcp and pxe boot for overcloud nodes
  - Later used as primary interface for ssh by InraRed (Ansible)
  - Data between compute nodes and Ceph storage (if exists)

  nic2 - management

  - Internal API for the overcloud services (services run REST queries against these interfaces (for example Neutron/Nova communication and neutron-server/neutron-agent communication))
  - Tenant network with tunnels (vxlan/gre/vlan) for internal data between OverCloud nodes. Examples:
    - VM (on compute-0) to VM (on compute-1)
    - VM (on compute-1) to Neutron Router (on Controller-3)

  nic3 - external

- public API for the overcloud services (OC users run REST queries against these interfaces)
- The testers (i.e. Tempest) use this network to execute commands against the OverCloud API
- Routes external traffic for nested VMs outside of the overcloud (connects to neutron external network and br-ex bridge...)
- The testers (i.e. Tempest) use this network to ssh to the VMs (cirros) nested in the OverCloud

To build a Virthost deployment, use the preset deployment-files provided in `settings`:

```
ir-installer ospd --deployment-files=$PWD/settings/installer/ospd/deployment/virt␣
↪[...]
```

InfraRed will generate `undercloud.conf` and `instackenv.json` configuration files if not provided explicitly. See Quickstart guide for more details.

Hostnames

To simplify node management, InfraRed uses shorter names than the default names OSPD gives the OverCloud nodes. For example, instead of `overcloud-cephstorage-0` the node will be called `ceph-0`. The full conversion details are here.

A user can provide customized HostnameMap using `--overcloud-hostname` argument:

```
ir-installer [...] ospd [...] --overcloud-hostname=special_hostnames.yml [...]
```

Listing 1.5: special_hostnames.yml

```
HostnameMap:
    ceph-0: my_main_ceph_node
    ceph-1: another_storage_node
    controller-2: SPECIAL_MACHINE
    compute-0: BIG_HYPERVISOR
```

Note that the default naming template is the one described above and not the one in the tripleo documentation (`overcloud-novacompute-0`).

Note: The naming convention and customization can be completely overridden if the `--deployment-files` input contains a file called `hostnames.yml` following the tripleo guidlines

Testers

Note: Inventory file (`hosts`) should have `tester` group with 1 node in it. In `ospd` this is usually the undercloud. In `packstack` this is usually a dedicated node.

For list of supported testers invoke:

```
$ ir-tester --help
```

Tempest

Note: *InfraRed* uses a python script to configure *Tempest*. Currently that script is only available in Red Hat's Tempest fork, so *InfraRed* will clone that repo as well in order to use that script.

Use `--tests` to provide a list of test sets to execute. Each test set is defined in settings tree And will be executed separately.

To import Tempest Plugins from external repos, `tests` files should contain `plugins` dict. *InfraRed* will clone those plugins from source and install them. Tempest will be able to discover and execute tests from those repos as well.

Listing 1.6: settings/tester/tempest/tests/neutron.yml

```
---
name: neutron
test_regex: "^neutron.tests.tempest"
whitelist: []
blacklist: []
plugins:
  neutron:
    repo: "https://github.com/openstack/neutron.git"
```

Scripts

Archive

This script will create a portable package which can be used to access an environment deployed by InfraRed from any machine. The archive script archives the relevant SSH & inventory files using tar. One can later use those files from anywhere in order to SSH and run playbook against the inventory hosts.

To get the full details on how to use the archive script invoke:

```
$ ir-archive --help
```

Basic usage of archive script:

```
$ ir-archive
```

Note: Unless supplying paths to all relevant files, please run this script from the InfraRed project dir

This creates a new tar file (IR-Archive-[date/suffix].tar) containing the files mentioned above while de-referencing local absolute paths of the SSH keys so they can be accessed from anywhere.

Usage examples:

- Untar the archive file:

  ```
  tar -xvf IR-Archive-2016-07-11_10-31-28.tar
  ```

Note: Make sure to extract the files into the InfraRed project dir

- Use the SSH config file to access your provisioned nodes:

```
ssh -F ansible.ssh.config.2016-07-11_10-31-28 controller-0
```

- Execute ansible Ad-Hoc command / Run playbook against the nodes in the archived inventory file:

```
ansible -i hosts-2016-07-11_10-31-28 all -m setup
```

- Use the archived files with InfraRed:

```
mv ansible.ssh.config.2016-07-11_10-31-28 ansible.ssh.config
ir-installer --inventory hosts-2016-07-11_10-31-28 ospd ...
```

Plugins

Plugins are essentially *Ansible* projects that use *InfraRed* to expose a predifined UI

Add new Plugins

There are two steps that should be done when adding a new plugin to InfraRed:

1. **Creating a specification file:** InfraRed uses ArgParse wrapper module called 'clg' in order to create a parser that based on *spec* file (YAML format file) containing the plugin options. The spec file should be named as the new plugin name with '.spec' extension and located inside the plugin dir under the InfraRed 'setting' dir. For more details on how to use this module, please see the Specifications documentation.
2. **Creating settings files.** Settings files are files containing data which defines how the end result of the playbook execution will be looked like. Settings file are file in YAML format, end with ".yml" extension. Those files located under the plugin's dir which itself located under the 'settings' dir in the InfraRed project's dir. The end result of the playbook execution is based on the data created by merging of several settings files together with other values, all are received by the user. When adding a new plugin, there is a need to create those settings files containing the needed data for the playbook execution.

Plugin Input

External setting trees

InfraRed builds settings tree (YAML dict-like structures) that are later passed to Ansible as varibales. This tree can be built upon pre-existing YAML files (with `-i/--input`) , or be overridden post creation by other pre-existing files and/or sets of `key=value` arguments.

The merging priority order is:

1. Input files
2. Settings dir based options
3. Extra Vars

InfraRed input arguments

InfraRed extends the `clg` and `argpars` packages with the following types that need to be defined in *.spec* files:

- **Value**: String values absolute path. For the argument name is "arg-name" and of subparser "SUBCOMMAND" of command "COMMAND", the default

- **YamlFile**: Expects path to YAML files. Will search for files in one of the configured settings directories before trying to resolve absolute path. If the argument name is "arg-name" and of subparser "SUBCOMMAND" of command "COMMAND", the default search path would be:

```
{settings_dir1,...,settings_dirN}/COMMAND/SUBCOMMAND/arg/name/arg_value
```

- **Topology**: Provisioners allow to dynamically define the provisioned nodes topology. InfraRed provides several 'mini' YAML files to describe different roles: `controller`, `compute`, `undercloud`, etc... These 'mini' files are then merged into one topology file according to the provided `--topology-nodes` argument value.

  The `--topology-nodes` argument can have the following format:

  - `--topology-nodes-controller:1,compute:1`
  - `--topology-nodes-controller:1`
  - `--topology-nodes-controller:3,compute:1,undercloud:1`

  InfraRed will read dynamic topology by following the next steps:

  1. Split the topology value with ','.
  2. Split each node with ':' and get pair (role, number). For every pair look for the topology folder (configured in the infrared.cfg file) for the appropriate mini file (controller.yml, compute.yml, etc). Load the role the defined number of times into the settings.

Note: The default search path for topology files is `{settings_dir(s)}/provivisioner/topology`. Users can add their own topology roles there and reference them on runtime

---

These arguments will accept input from sources in the following priority order:

1. Command line arguments: `ir-provision virsh --host-address=some.host.com --host-user=root`
2. Environment variables: `HOST_ADRRESS=earth.example.com ir-provision virsh --host-user=root`
3. Predefined arguments in ini file specified using `--from-file` option:

```
ir-provision virsh --host-address=some.host.com --from-file=user.ini

cat user.ini
[virsh]
host-user=root
host-key=mkey.pm
```

Note: Do not use double quotes or apostrophes for the string values in the configuration ini file. Infrared will NOT remove those quotation marks that surround the values.

---

1. Defaults defined in `.spec` file for each argument.

Note: The sample *ini* file with the default values can be generated with: `ir-povision virsh --generate-conf-file=virsh.ini`. Generated file will contain all the default arguments values defined in the spec file.

Arguments of the above types will be automatically injected into settings YAML tree in a nested dict from.

Example: The input for `ir-COMMAND` and argument `--arg-name=arg-value` maps to:

```
COMMAND:
    arg:
        name: "arg-value"
```

"arg-value" can be a simple string or be resolved into a more advanced dictionary depending on the argument type in `.spec` file

Extra-Vars

Set/overwrite settings in the output file using the '-e/–extra-vars' option. There are 2 ways of doing so:

1. **Specific settings: (`key-value` form)** `-e provisioner.site.user-a_user`
2. **Path to a settings file: (starts with `@`)** `-e @path/to/a/settings_file.yml`

The `-e`/`--extra-vars` can be used more than once.

Advanced features

Tags

Advanced usage sometimes requires partial execution of the ospd playbook. This can be achieved with Ansible tags

List the available tags of the *ospd* playbooks:

```
ir-installer [...] ospd [...] --ansible-args list-tags
```

Execute only the desired tags. For example, this will only install the UnderCloud and download OverCloud images:

```
ir-installer [...] ospd [...] --ansible-args "tags=undercloud,images"
```

Breakpoints

Commonly used tags:

undercloud Install the UnderCloud.

images Download OverCloud images and upload them to UnderCloud's Glance service.

introspection Create `instackenv.json` file and perform introspection on OverCloud nodes with Ironic.

tagging Tag Ironic nodes with OverCloud properties.

overcloud_init Generate heat-templates from user provided `deployement-files` and from input data. Create the `overcloud_deploy.sh` accordingly.

overcloud_deploy Execute `overcloud_deploy.sh` script

overcloud Do `overcloud_init` and `overcloud_deploy`.

inventory_update Update Ansible inventory and SSH tunneling with new OverCloud nodes details (user, password, keys, etc...)

Common use case of tags is to stop after a certain stage is completed. To do this, Ansible requires **a list of all the tags up to, and including the last desired stage**. Therefore, in order to stop after UnderCloud is ready:

```
ir-installer [...] ospd [...] --ansible-args --ansible-args tags="init,dump_facts,
↪undercloud"
```

Or, in as a bash script, to stop after `$BREAKPOINT`:

```
FULL_TAG_LIST=init,dump_facts,undercloud,virthost,images,introspection,tagging,
↪overcloud,inventory_update
LEADING=`echo $FULL_TAG_LIST | awk -F$BREAKPOINT '{print $1}'`
ir-installer [...] ospd [...] --ansible-args --ansible-args tags=${LEADING}$
↪{BREAKPOINT}
```

OverCloud Image Update

OSPD creates the OverCloud nodes from images. These images should be recreated on any new core build. However, this is not always the case. To updates that image's packages (after download and before deploying the Overcloud), to match RH-OSP core bits `build`, set `--images-update` to `yes`:

```
ir-installer [...] ospd [...] --images-update=yes
```

Note: This might take a while and sometimes hangs. Probably due to old libguestfs packages in RHEL 7.2. For a more detailed console output of that task, set `--images-update` to `verbose`.

Custom repositories

Infrared allows to add custom repositories to the UnderCloud when you're running *OSPD*, after installing the default repositories of the *OSPD* release. This can be done passing through `--extra-vars` with the following key:

- `ospd.extra_repos.from_url` which will download a repo file to `/etc/yum.repos.d`

1. Using `ospd.extra_repos.from_url`:

   Create a yaml file:

   Listing 1.7: repos.yml

   ```
   ---
   installer:
      extra_repos:
          from_url:
              - http://yoururl.com/repofile1.repo
              - http://yoururl.com/repofile2.repo
   ```

   Run ir-installer:

   ```
   ir-installer --extra-vars=@repos.yml ospd
   ```

1. Using ospd.extra_repos.from_config

   Using this option enables you to set specific options for each repository:

Listing 1.8: repos.yml

```
---
installer:
   extra_repos:
       from_config:
           - { name: my_repo1, file: my_repo1.file, description: my␣
↪repo1, base_url: http://myurl.com/my_repo1, enabled: 0, gpg_check: 0 }
           - { name: my_repo2, file: my_repo2.file, description: my␣
↪repo2, base_url: http://myurl.com/my_repo2, enabled: 0, gpg_check: 0 }
...
```

Note: As you can see, ospd.extra_repos.explicity support some of the options found in yum_repository module (name, file, description, base_url, enabled and gpg_check). For more information about this module, visit Ansible yum_repository documentation.

Run ir-installer:

```
ir-installer -e @repos.yml ospd
```

Custom/local tempest tester

You might have a specific version of tempest to test locally in a particular directory, and you want to use it. Infrared allows you to use this instead of the default git repository. To do so, all you need to do is pass the key tester.local_dir as extra-vars to ir-tester:

Run ir-tester:

```
ir-tester tempest --extra-vars tester.local_dir=/patch/for/your/tempest
```

Scalability

Infrared allows to perform scale tests on different services.

Currently supported services for tests:

- compute
- ceph-storage
- swift-storage

1. To scale compute service:

   Deployment should have at least 3 compute nodes.

   Run ansible playbook:

   ```
   ansible-playbook -vvvv -i hosts -e @install.yml playbooks/installer/ospd/
   ↪post_install/scale_compute.yml
   ```

   It will scale compute nodes down to 1 and after that scale compute node back to 3.

2. To scale ceph-storage service:

Deployment should have at least 3 ceph-storage nodes.

Run ansible playbook:

```
ansible-playbook -vvvv -i hosts -e @install.yml playbooks/installer/ospd/
↪post_install/ceph_compute.yml
```

It will scale compute nodes down to 1 and after that scale compute node back to 3.

3. To scale swift-storage service:

   Deployment should have at least 3 swift-storage nodes.

   Run ansible playbook:

```
ansible-playbook -vvvv -i hosts -e @install.yml playbooks/installer/ospd/
↪post_install/swift_compute.yml
```

Note: Swift has a parameter called `min_part_hours` which configures amount of time that should be passed between two rebalance processes. In real production environment this parameter is used to reduce network load. During the deployment of swift cluster for further scale testing we set it to 0 to decrease amount of time for scale.

UnderCloud testing

Usually, all tempest tests are run from the UnderCloud, against OverCloud, while you might want test UnderCloud services (e.g. ironic). The following cookbook uses InfraRed to run Tempest tests against the UnderCloud.

1. **We want an explicit "tester" node to avoid running tests on the same node as the UnderCloud.** Use "ironic" node instead of "undercloud". It's the same but doesn't have the role of "tester". Rename "controller" node into "test-vm" to avoid misunderstanding and update it's parameters to match with "baremetal" flavor.:

```
ir-provisioner -d virsh -v -o provision.yml \
    --topology-nodes=ironic:1,controller:1,tester:1 \
    --host-address=$HOST \
    --host-key=$HOME/.ssh/rhos-jenkins/id_rsa \
    --image=$IMAGE \
    -e @private.yml \
    -e provisioner.topology.nodes.controller.cpu=1 \
    -e provisioner.topology.nodes.controller.disks.disk1.size=41G \
    -e provisioner.topology.nodes.controller.memory=4096 \
    -e provisioner.topology.nodes.controller.name=test-vm
```

2. **As we don't want the full OSPD installation, we will use explicit *Tags* to do only certain parts:**

   - Undercloud - will install UnderCloud
   - Images - installs or builds OverCloud images
   - Ironic - performs all required actions before introspection (including assignment of the kernel and ramdisk)
   - Virthost - enables "virthost" specific tasks in case of "virsh" provisioning:

```
ir-installer --debug ospd -v --inventory hosts \
    -e @provision.yml \
    -e @private.yml \
```

```
    -o install.yml \
    --deployment-files=$PWD/settings/installer/ospd/deployment/virt \
    --product-version=10 \
    --product-core-version=10 \
    --ansible-args="tags=undercloud,images,virthost,ironic"
```

3. **We want prepare environment for ironic tests:**
    - update baremetal flavor with cpu_arch
    - create initial tempest.conf file using predefined template
    - enable ironic inspector
    - enable fake and pxe_ssh drivers in ironic
    - make desired neutron network shared
    - install rhos-release repos into "tester" node
    - configure data network on "tester" node:

```
ansible-playbook -vvvv -i hosts -e @install.yml \
    playbooks/installer/ospd/post_install/add_nodes_to_ironic_list.yml \
    -e net_name=ctlplane \
    -e driver_type=pxe_ssh \
    -e rc_file_name=stackrc
```

4. **Finally run the Ironic tempest plugin tests:: ** Run ir-tester:

```
ir-tester --debug tempest -v \
    -e @install.yml \
    --tests=ironic_inspector \
    -o test.yml
```

Virthost packages/repo requirements

Virsh

UEFI mode related binaries

According to usage UEFI with QEMU there is only one way to get the UEFI mode boot working with VMs, that often requires by Ironic team due to lack of hardware or impossibility to automate mode switching on baremetal nodes.

1. Add repo with OVMF binaries:

```
yum-config-manager --add-repo http://www.kraxel.org/repos/firmware.repo
```

2. Install OVMF binaries:

```
yum install -y edk2.git-ovmf-x64
```

3. Update QEMU config adding the following to the end of the /etc/libvirt/qemu.conf file:

```
nvram = [
    "/usr/share/edk2.git/ovmf-x64/OVMF_CODE-pure-efi.fd:/usr/share/edk2.git/ovmf-
↪x64/OVMF_VARS-pure-efi.fd"
]
```

4. Restart libvirt service:

```
systemctl restart libvirtd
```

Specifications

InfraRed "drives" Ansible through a Plugin's playbooks (and roles) in the following manner:

```
ir-XXXer YYYer [...]
```

Where `XXX` is the *command* (`provision`, `install`, or `test`), and `YYY` is the plugin *subcommand* (`virsh`, `ospd`, `openstack`, `tempest`, etc...)

- Each *command* executes a matching playbook (at `playbooks/XXX.yml`) with a generated set of extra vars as *plugin input*.
- That "top" playbook calls (via "include") to the *subcommand*'s playbook at `playbooks/XXXer/YYY.yml`

Plugin Input

InfraRed exposes several types of arguments via it's CLI to accept user-input before execution. It generates a python-dict input and merges it with a dict of defaults defined in YAML format.

If the *subcommand* is called `YYY`, InfraRed will search for its input definitions in settings trees in a directory called `YYY`.

Infrared uses special files (in YAML format) to describe plugin CLI interface. These files are called *specifications* (spec's) and have `.spec` extension.

The main idea of specification is to describe:

- all the possible options we can pass to the plugin
- any default values for the options
- required and optional options
- limitation for certain options, like choosing option value from the predefined list of allowed values

Infrared parses and merges all the spec files under the settings folders and pass all the defined options to the `argparse` module which is then used for cli options parsing.

Specification parser is derived from 'clg' module homepage.

Commands and subcommands

Infrared uses the positional arguments (subcommands) to extend functionality for the ir-* cli commands.

```
ir-provisioner [..] openstack [...]
   ^----------^       ^--------^
     command          subcommand
```

For example, the provisioner command aggregates several subcommands which define specific provisioners like virsh, openstack, beaker, foreman, etc.

The command specification files are stored under the `settings/<command_name>/` folders.

Command specification should start from the root of the spec file without any additional keywords:

```
---
options: [....]
groups:  [....]
```

All the subcommand specifications files are stored under the `settings/<command_name>/<subcommand_name>` folders.

Subcommands can be defined with the `subparsers` keyword followed by the subcommand name:

```
---
subparsers:
    virsh:
        options:
        [....]
        groups:
        [....]
```

It's recommended to define subcommands in the seprate .spec file.

Infrared settings structure

```
[settings]
    |
    +-> [installer]
    |       |
    |       +-> [ospd]
    |       |       |
    |       |       +-> ospd.spec
    |       |       |
    |       |       +-> ospd.yml
    |       |
    |       +-> [packstack]
    |       |       |
    |       |       +-> packstack.spec
    |       |       |
    |       |       +-> packstack.yml
    |       |
    |       +-> installer.spec
    |
    +-> [provisioner]
    |       |
    |       +-> [....]
    |       |
    |       +-> provisioner.spec
    |       |
    |       +-> provisioner.yml
    |
    +-> base.spec
```

The `base.spec` file contains:

- groups and options common for all the commands
- reusable groups (shared_groups)

The command specification files `installer/installer.spec` and `provisioner/provisioner.spec` contain:

- specific options and groups for a given command. For example, by default ir-provisioner command has the –debug flag to debug information into console.

The subcommand specification files `installer/ospd/ospd.spec` and `installer/ospd/packstack.spec` contain:

- subcommand name and description
- specific options and groups for a given subcommand

The subcommand default files `installer/ospd/ospd.yml` and `installer/ospd/packstack.yml` contain:

- A set of extra vars in YAML format which the subcommand will use as the skeleton for its input

Options and Groups

An option can be defined with an `options` keyword followed by the dict of options. Every key in that dict is an option name, and value is another dict of option parameters.

```
---
options:
    debug:
        help: Run InfraRed in DEBUG mode
        short: d
        action: store_true

    verbose:
        help: Control Ansible verbosity level
        short: v
        action: count
        default: 0
```

Infrared transforms that to the CLI tool with the following arguments:

```
ir-command [-h] [-d] [-v]

optional arguments:
-h, --help            show this help message and exit
-d, --debug           Run InfraRed in DEBUG mode
-v, --verbose         Control Ansible verbosity level
```

Options configuration

Every option in the specification can have the following keywords:

- *short* (`infrared`)
- *help* (`argparse`)
- *required* (`argparse`)
- *default* (`argparse`)
- *choices* (`argparse`)
- *action* (`argparse`)
- *nargs* (`argparse`)

- *const* (`argparse`)
- *type* (`argparse`)
- *silent* (`infrared`)
- *required_when* (`infrared`)

short

This section must contain a single letter defining the short name (beginning with a single dash) of the current option.

help

argparse link: https://docs.python.org/dev/library/argparse.html#help

A brief description of what the argument does.

required

argparse link: https://docs.python.org/dev/library/argparse.html#required

Whether or not the command-line option may be omitted.

type

argparse link: https://docs.python.org/dev/library/argparse.html#type

The type to which the command-line argument should be converted.

There are two groups of type supported by Infrared:

- control types: all the builtin types such as 'str', 'int' and other. Option with these types are used to control Infrared behavior and will not be put into the generated settings files. For example, ir-provisioner command has 'debug' control option.
- settings types (Value types): `Value`, `YamlFile`, `Topology` and other types. Options with these types will be put by Infrared into the settings files.

If type is not specified, Infrared will treat such option as 'str' control option.

Settings types

- *Value*
- *YamlFile*
- *ListOfYamls*
- *Topology*
- *DictValue*

Value

Simple value which will be put into the command settings. For example if for 'provisioner' command and the 'virsh' subcommand with options:

```
---
subparsers:
    virsh:
        options:
            host-address:
                type: Value
                help: 'Address/FQDN of the BM hypervisor'
                required: yes
```

Calling the 'ir-provisioner' cli tool:

```
ir-provisioner virsh --host-address myhost.domain.com
```

will produce the folloiwng settings in YAML format:

```
---
provisioner
    host:
        address: myhost.domain.com
```

This settings tree is passed to Ansible as extra-vars.

YamlFile

Loads the content of the specified YAML file into the settings. For the option named 'arg-name' Infrared will look for YAML file into the following locations:

1. <settings folder>/<command name>/<subcommand name>/arg/name/<file_name>
2. <settings folder>/<command name>/arg/name/<file_name>
3. ./arg/name/<file_name>

For example, the 'provisioner' command and virsh 'subcommand' has the YamlFile option:

```
---
subparsers:
    virsh:
        options:
            topology-network:
                type: YamlFile
    ....
```

Command call:

```
ir-provisioner virsh --topology-network=default.yml
```

Infrared will look for *default.yml* in the following locations:

1. settings/provisioner/virsh/topology/network/default.yml
2. settings/provisioner/topology/network/default.yml
3. ./topology/network/default.yml

Content of the *default.yml* will be put into the settings file:

```
---
provisioner:
    topology:
        network:
            # content of the default.yml will go there
            key1: value
            key2: value
            ....
```

Topology

Topology type is used to describe what nodes (vm's) should be provisioned by the provisioner.

Topology value should be the list of nodes names and the number of nodes: `<node name>:<node number>, <node2 name>:<node2 number>,...`. For example:

```
ir-provisioner virsh --topology-nodes=undercloud:1,controller:2,compute:3
ir-provisioner virsh --topology-nodes=controller:3
```

Every node name maps to the appropriate YAML file (undercloud.yml. controller.yml, controller.yml) that should be stored in one the following locations:

1. <settings folder>/<command name>/<subcommand name>/arg/name/<file_name>
2. <settings folder>/<command name>/arg/name/<file_name>
3. <settings folder>/<command name>/topology/<file_name>
4. ./arg/name/<file_name>

All the YAML files will be loaded into the settings under the node name key. 'Amount' key will be adjusted.

For example, for `undercloud:1,controller:2,compute:3` value with option name `topology-nodes` the settings file will be:

```
---
provisioner:
    topology:
        nodes:
            undercloud:
                # content of the undercloud.yml will go there
                amount: 1
             controller:
                # content of the controller.yml will go there
                amount: 2
             compute:
                # content of the compute.yml will go there
                amount: 3
```

ListOfYamls

Specifies the list of YAML files to load into the settings.

Option value should be the comma separated string of files to load with or without yml extension. Single element in list is also accepted.

Values examples:

- item1,item2,item3
- item1.yml

Search locations are the same as for the `YamlFile` type.

For example, for `network,compute,volume` value with option name `tests`, command `tester` and subcommand `tempest`, the settings file will be:

```
---
tester:
    tests:
        network:
            # content of the network.yml will go there

        compute:
            # content of the compute.yml will go there

        volume:
            # content of the volume.yml will go there
```

DictValue

Specifies the value which should be interpreted as a dictionary value in the settings.

DictValue should be specified in the format: `option1=value1;option2=value;option3=value3`

Consider the following example on how to add the DictValue option into a spec.

```
---
subparsers:
    virsh:
        options:
            my-dict-option:
                type: DictValue
                help: 'Sample dict'
```

Calling the cli tool:

```
ir-provisioner virsh --my-dict-option=option1=value1;key2=value2
```

will produce the following dict tree in YAML format:

```
---
provisioner
    my:
        dict:
            options:
                option1: value1
                key2: value2
```

This settings tree is passed to Ansible as extra-vars.

Types extension

Settings types can be extended by adding user class to the `clg.COMPLEX_TYPES dictionary`. Complex types should implement the `clg.ComplexType` interface:

```
import clg
from datetime import datetime

class DateValue(ComplexType):

    def resolve(self, value):
        try:
            return datetime.strptime(value, '%d/%m/%Y')
        except Exception as err:
            raise clg.argparse.ArgumentTypeError(err)

COMPLEX_TYPES['DateValue'] = DateValue

# proceed with clg usage
...
```

YAML configuration is then can look like:

```
---
options:
    date:
        help: Date value
        type: DateValue
...
```

Control types can be extended by adding callable objects which accept one argument (value) to the `clg.TYPES` dictionary.

default

argparse link: https://docs.python.org/dev/library/argparse.html#default

The value produced if the argument is absent from the command line.

choices

argparse link: https://docs.python.org/dev/library/argparse.html#choices

A container of the allowable values for the argument.

action

argparse link: https://docs.python.org/dev/library/argparse.html#action

The basic type of action to be taken when this argument is encountered at the command line.

Infrared provides two actions which allows to read options from INI files and generate simple configuration files.

```
---
options:
    from-file:
        action: read-config
        help: reads arguments from file.
    generate-conf-file:
        action: generate-config
        help: generate configuration file with default values
```

nargs

argparse link: https://docs.python.org/dev/library/argparse.html#nargs

The number of command-line arguments that should be consumed.

const

argparse link: https://docs.python.org/dev/library/argparse.html#const

Value in the resulted *Namespace* if the option is not set in the command-line (*None* by default).

silent

Specifies which required arguments should become no longer required when this option is set.

```
---
options:
    image:
        type: YamlFile
        help: 'The image to use for nodes provisioning. Check the "sample.yml.example
↪" for example.'
        required: yes
    ...
    cleanup:
        action: store_true
        help: Clean given system instead of running playbooks on a new one.
        silent:
            - "image"
...
```

In that example the image will no longer be required when cleanup option is set.

required_when

Specifies condition when options should became required.

Condition should be specified in form <*option_name*> == <*value*>.

```
---
options:
    images-task:
        type: Value
        choices: [import, build, rpm]
```

```
        default: rpm

    images-url:
        type: Value
        help: Specifies the import image url. Required only when images task is
↪'import'
        required_when: "images-task == import"
```

Groups

If options belong to one area or connected somehow, they can be grouped:

```
---
groups:
    - title: Hypervisor
      options:
          host-address:
              type: Value
              help: 'Address/FQDN of the BM hypervisor'
              required: yes
          host-user:
              type: Value
              help: 'User to SSH to the host with'
              default: root
          host-key:
              type: Value
              help: "User's SSH key"
              required: yes
```

Shared groups

Shared groups allow to include predefined options groups into different commands or subcommands

Shared groups should be defined in the `settings/base.spec` file or in the command spec file:

```
---
shared_groups:
    - title: Inventory hosts options
      options:
        inventory:
            help: Inventory file
            type: str
            default: hosts

    - title: Common options
      options:
        dry-run:
            action: store_true
            help: Only generate settings, skip the playbook execution stage
        input:
            action: append
            type: str
            short: i
            help: Input settings file to be loaded before the merging of user args
```

Shared group can be included into the **command** spec file with the `include_groups` directive:

```
---
include_groups: ["Debug Options"]
```

For a **subcommand** the `include_groups` should be defined under the subparsers section:

```
---
subparsers:
    virsh:
        include_groups: ["Ansible options", "Inventory options", "Common options",
↪"Configuration file options"]
```

Options sources

Infrared is not limited with the CLI options only. We can pass arguments to the plugin using the following approaches:

- through the CLI options
- through INI files using the `--from-file` argument or any other argument with `action: read-config` attribute in specification
- through environment variables

Infrared resolves option value in the next order:

1. If option value is provided by CLI, use that value.
2. Else use value from INI file if it is defined there.
3. Else use environment variable (with the same name as an option name, but capitalized and '-' replaced with '_' (for example, 'arg-name' will be transformed to ARG_NAME env variable).
4. Else use value specified by the default keyword in the spec file.
5. If default value is not specified, option will not be defined.

Consider the following subcommand specification as an example:

```
---
subparsers:
    testcommand:
        groups:
            - title: common options
              options:
                  from-file:
                      action: read-config
                      help: reads arguments from file.


            - title: test options
              options:
                  option1:
                      type: Value
                  option2:
                      type: Value
                  option3:
                      type: Value
```

The INI file with the settings:

```
[testcommand]
option1=ini_value1
option2=ini_value2
```

Invoke subcommand with the following options:

```
OPTION2=env_value2 OPTION3=env_value3 ir-somecomand testcommand --from-file=test.ini -
↪-option1=cli_value1
```

This will produce the follwing arguments:

- option1 = cli_value1
- option2 = ini_value2
- option3 = env_value3

Contact Us

Team:

Tal Kammer	tkammer@redhat.com
Yair Fried	yfried@redhat.com

GitHub:

Issues are tracked via GitHub. For any concern, please create a new issue.

Contributors Guide

Sending patches

Changes to project are accepted via review.gerrithub.io. For that you need to be member of our group rhosqeauto-core on gerrithub, ask any of the current members about it.

You can use git-review (dnf/yum/pip install). To initalize in the directory of InfraRed execute `git review -s`. Every patch needs to have *Change-Id* in commit message (git review -s installs post-commit hook to automatically add one).

For some more info about git review usage, read GerritHub Intro and OpenStack Infra Manual.

Release Notes

v1.1.0

New Features

- Added support for OSPD on Bare-Metal machines (documentation pending).
- Move to GerritHub

- Improve Documentaion
- Unit-Testing via tox
- **OSPD**
  - Internal Swift storage backend
  - Older OverCloud versions with New UnderCloud (OSP-d #8 and above). For example: Deploy OverCloud of OSP #7 with OPS-d #8 UnderCloud
- **Scale**
  - Internal Ceph
  - Internal Swift
  - Compute
- Collect logs - ansible playbook to grab required data and logs from all nodes post run. Allows to debug the setup even after it was destroyed:

```
ansible-playbook -i hosts -e @SETTINGS_YAML_FILE playbooks/collect-logs.yml
```

  Jobs can now ship logs to Logstash cluster
- Help for arguments of type `YamlFileArgumet` lists available files from default locations.
- Reprovision via Foreman and IPMI
- Reprovision and reserve via Beaker
- Configure multiple `settings` trees. Will look for file arguments in multiple settings directories as listed in `infrared.cfg`
- **Generate better config files:**
  - Put all the required arguments to the generated config ini file
  - If default value is not provided - put the placeholder for that parameter in ini file
  - Resolve only current spec arguments.
  - Infrared allows to use ir-* command in two steps:

    ir-* –generate-conf-file=file.ini ir-* –from-file=file.ini
- Use existing image snapshots with `virsh` provisioner (faster than building the images)
- `openstack` provisioner accepts private DNS server address.
- Add Ansible tags to `ospd` workflow so advanced users can ivoke partial ospd tasks (`undercloud`, `introspection`, `overcloud`, etc...)
- Add Tempest tester:

  ir-tester tempest –help
- Customized hostnames for controller nodes
- Adds support for OSP #10
- OSPD post-install actions no longer invoked during `ir-installer ospd` run. Need to be explictly invoked via advanced Ansible call:

```
ansible-playbook -i hosts -e @SETTINGS_YAML_FILE playbooks/installer/ospd/post_
↪install/ACTION.yml
```

- Configure fencing of overcloud nodes (virsh only) with post-install playbook.
- Invetory files created for each invocation (`hosts-provisioner` and `hosts-installer` are created, instead of overwriting the same `hosts-$USER` file.)

Bug Fixes

- SSH to OverCloud nodes: OSPD reprovisions OverCloud machines with new addresses and credentials. Final stage of install uses built-in openstack module to get OverCloud info from UnderCloud (`nova list`) and recreate invetory and ssh config files.
- **Version conflicts:**
  - pin `Babel`
  - Removed `configure` module
  - Blacklist `Ansible` 2.1.0
  - pin `shade`
- Default config file is up to date
- **Packstack:**
  - Added All-In-One (`aio.yml`) topology support
  - Fixed network tasks on controller (No longer support dedicated network nodes)
- Collect Logs: Avoid archiving virsh machines on `virthost` node.
- Improve lookup: No longer fails when there are multiple visits to the same key in the lookup
- Faster lookup with unittest.
- `virsh` provisioner no longer fails if `sshpass` is not installed
- Remove "sample" files from genertad config files.
- Resolve ~ (expanduser) on `extra-vars` file input (`--extra-vars @~/my/file`)
- Informative failure message for bad topology syntax
- Single default inventory file for all `ir-*` tools
- Beaker - Proper Ansible failure message when `ca_cert` file is missing
- Remove empty placeholer file for rhos-8.0 workarounds
- `openstack` provisioner no longer registers the same IP address for instances of the same node
- Fix internal ceph backend: `glance image-create` no longer fails with ceph backend
- Fix merging lists in inpute files.
- `rhos-release` should pin latest version
- Verify that `overcloudrc` file is created after overcloud deployment succeeds
- Install python-virtualenv on the undercloud (required for `shade`)
- Add ipv6 support for virsh external network
- Cast the string value of `product` to int

CHAPTER 2

Indices and tables

- genindex
- modindex
- search